TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems

Yuanjiang Ni*
Alibaba Group
USA

Pankaj Mehra
Elephance Memory, Inc.
USA

Ethan Miller
UC Santa Cruz
Pure Storage
USA

Heiner Litz
UC Santa Cruz
USA

Abstract

Main memory dominates data center server cost, and hence data center operators are exploring alternative technologies such as CXL-attached and persistent memory to improve cost without jeopardizing performance. Introducing multiple tiers of memory introduces new challenges, such as selecting the appropriate memory configuration for a given workload mix. In particular, we observe that inefficient configurations increase cost by up to 2.6× for clients, and resource stranding increases cost by 2.2× for cloud operators. To address this challenge, we introduce TMC, a system for recommending cloud configurations according to workload characteristics and the dynamic resource utilization of a cluster. Whereas prior work utilized extensive simulation or costly machine learning techniques, incurring significant search costs, our approach profiles applications to reveal internal properties that lead to fast and accurate performance estimations. Our novel configuration-selection algorithm incorporates a new heuristic, packing penalty, to ensure that recommended configurations will also achieve good resource efficiency. Our experiments demonstrate that TMC reduces the search cost by up to 4× over the state-of-the-art, while improving resource utilization by up to 17% as compared to a naive policy that requests optimal tiered memory allocations in isolation.

*Work was performed as a graduate student at UC Santa Cruz

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624667

CCS Concepts

• **Information systems** → *Storage class memory*; • **Software and its engineering** → **Memory management**.

Keywords

Tiered memory management, Resource allocation

ACM Reference Format:
Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. 2023. TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620678.3624667

1 Introduction

Continuing growth in data center applications' main memory requirements [9, 50, 76], along with the slowdown of DRAM scaling, renders main memory one of the costliest components of data center infrastructure. Moreover, due to the strict service level objectives (SLO) imposed by data center applications, memory is often over-provisioned. Alibaba reported [31] CPU and memory utilization of 38% and 88%, respectively, in their clusters, while Microsoft found [45] that 50% of the provisioned main memory capacity remains untouched by virtual machines (VM). Emerging storage class memories (SCM) [1, 14, 23, 38, 39, 42, 46, 77, 78] promise higher density and lower energy cost while only moderately degrading performance. Memory disaggregation [4, 30, 47, 48, 76] and Compute Express Link (CXL) attached memory [16, 45, 50, 64] enable pooled deployment of recycled, slower, previous-generation memory across a fabric. While remote memory induces higher access latency, it significantly reduces costs, as shown in prior work [4, 40, 45]. Disaggregated, CXL-attached, and persistent memories are practical, as they can be mapped into application virtual address space and accessed using conventional load-store instructions. Although such tiered-memory systems are now increasingly supported by operating systems such as Linux [39, 50, 76, 78], it remains unclear how applications should allocate their data

structures between the faster, more expensive and the slower, less-expensive tiers for maximizing performance per total cost of ownership (TCO).

Cloud providers such as Microsoft Azure [53], Amazon AWS [5], and Google [28] offer Infrastructure as a Service (IaaS) to satisfy the computing needs of their clients. In such platforms, optimizing tiered memory systems becomes even more challenging as the optimal allocation of memory now depends on multiple applications and their requirements. The selection of the right resources benefits the cloud provider and client. It reduces the overall computation cost, informs pricing models, and enables providers to build physical systems with the right amount of DRAM and alternative memory technologies.

For instance, as we will show in Section 2.1, a wrong configuration increases the TCO by up to 2.6× for the cloud customer. In addition, the optimization of cloud configurations also determines the overall resource efficiency of a data center. In particular, to reduce resource stranding, an efficient tiered-memory configuration policy also needs to consider the actual available amount of memory in the different tiers.

Optimizing the cloud resource configurations such as the fast to slow memory ratio in tiered memory systems and predicting its performance implications is challenging due to the large search space. Utilizing a brute-force approach, an infeasible number of configurations covering all fast to slow memory ratios and hardware resources needs to be evaluated to devise accurate performance and TCO models. Prior works such as Cherrypick [3] and Selecta [41] have tried to address this challenge by obtaining end-to-end performance measurements of a considerable number of configurations learning the application's sensitivity to memory bandwidth and latency. These systems utilize Bayesian Optimization (BO) and Collaborative Filtering (CF) to reduce the search space, by predicting the performance of configurations based on a small set of measurements. Nevertheless, these techniques suffer from several shortcomings. BO, for instance, does not generate a performance model, and hence it is unable to explore Pareto-optimal cost-performance trade-offs required to minimize the cost for both the cloud provider and customer (*i.e.* it predicts only a single performance-optimal configuration). CF [20, 41] has been applied to reduce search overheads, however, its complexity still scales linearly with the number of explored configurations while also failing to provide a performance model.

To overcome these challenges, we introduce the tiered memory configurator (TMC), a mechanism to effectively manage systems with multiple tiers of memory, including the last-level cache (LLC), slow memory (SCM/CXL/remote), and the fast memory (DRAM) tier. TMC recommends near-optimal tiered-memory configurations according to the behavior of the application and the real-time utilization of the data center. Instead of utilizing a machine-learning-based, black-box approach as in prior works, TMC devises a performance model based on the understanding of hardware performance characteristics. Our resulting model only contains a minimal amount of workload-specific variables, which only requires three profile runs of an application. This separates our work from techniques such as simple regression [54], which requires many profile runs to obtain a predictive model. To optimize both performance and cost, our methodology optimizes for a single metric: performance per TCO. In particular, we determine the additional hardware resources that are required to offset a performance degradation to compute the holistic performance per TCO of a system. In addition, we introduce a new heuristic, *packing penalty*, which penalizes the configurations that lead to resource-stranding. As a result, TMC not only optimizes performance per TCO for a particular user but also ensures efficient real-time resource utilization for the data center operator. In summary, this paper makes the following contributions:

- We investigate hardware-based profiling that can be used to unearth the application-specific properties for estimating performance per TCO.
- We propose a comprehensive memory performance estimation technique that requires only three profile runs.
- We propose a novel optimization mechanism that produces ideal configurations for both cloud customers and providers.
- We show that TMC reduces the search overhead by 3× compared to the state-of-the-art improving resource efficiency in the cloud by 17%.

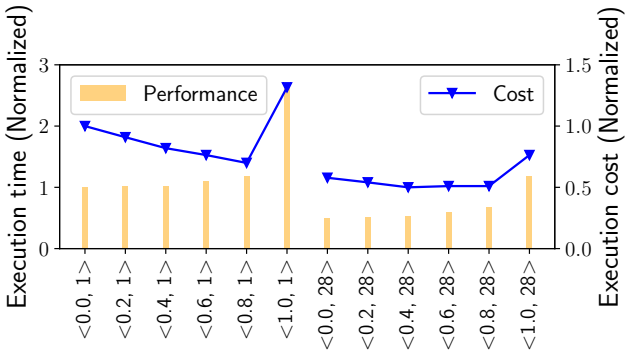


Figure 1: Execution time and cost analysis for 12 ⟨Slow memory ratio, LLC capacity⟩ configurations (*graph500* workload): The slow memory ratio represents the fraction of slow memory in relation to the total memory capacity. The LLC capacity indicates the number of LLC ways allocated in each configuration.

2 Background and Motivation

In this section, we discuss the application of multiple memory tiers in the datacenter. Then we discuss how the selection of the VM's memory configuration affects resource efficiency.

2.1 Cost Efficiency for Cloud Customers

Public IaaS cloud providers such as Amazon's EC2 [5] offer their customers a limited number of predefined VM instance types and charge them on a per-hour basis. On the other hand, a few IaaS cloud providers, such as Google's Compute Engine, further allow cloud users to create a VM instance with a customized number of vCPUs and amount of memory [28]. Prior work [84] has shown that enabling VM customization is beneficial for both the provider and the user. This work targets a future IaaS cloud incorporating both traditional DRAM and additional, slower memory tiers. We expect, that in future clouds, customers will be able to configure the number of vCPUs, the amount of local DRAM, the amount of second-tier memory, and the capacity of the last-level cache (LLC) of their vCPUs in a VM. Cloud providers have the option to leverage technologies like Intel's cache allocation technology (CAT)[34, 65] or fine-grain cache partitioning techniques such as vantage[65] to dynamically adjust the sizes of the last-level caches for different applications. The IaaS cloud then charges the users based on Equation 1.

$$Total\ cost = VM\ cost \times Execution\ time \quad (1)$$

Scaling down a VM configuration reduces its hourly VM cost, however, it also increases the execution time of applications. To minimize the total cost, one must choose a proper configuration that optimizes both. Figure 1 shows the execution time and cost for *graph500* utilizing various configurations with different allocations of slow and fast memory as well as different last-level cache sizes. As the fraction of slow memory increases, the total cost is reduced while the run time is increased. This is because the hourly cost savings outweigh the performance penalty of scaling down the VM. However, the total cost eventually increases as the higher run time exceeds the savings in hourly cost. Note that in this experiment, we translate a performance slowdown into cost by computing the additional total number of VMs required to offset the performance degradation. This assumes that applications are throughput-bound, *i.e.* an increase in execution time can be offset with additional hardware resources. This assumption is typical for data centers that scale user request-level throughput with hardware resources or deploy high fan-out architectures (*e.g.* Google Websearch) to distribute the execution time of a request across servers.

Applications	Config 1	Config 2	Config 3
	2× slower, 0.7× cost	3× slower, 0.4× cost	4× slower 0.3× cost
cactusBSSN	⟨ 0.1, 28 ⟩	⟨ 0.1, 28 ⟩	⟨ 1.0, 28 ⟩
graph500	⟨ 0.4, 28 ⟩	⟨ 0.4, 28 ⟩	⟨ 0.9, 20 ⟩
memcached	⟨ 0.0, 2 ⟩	⟨ 0.9, 2 ⟩	⟨ 0.9, 2 ⟩
xsbench	⟨ 0.9, 4 ⟩	⟨ 0.9, 3 ⟩	⟨ 0.9, 3 ⟩
canneal	⟨ 0.6, 2 ⟩	⟨ 0.6, 1 ⟩	⟨ 0.6, 1 ⟩
xhpcg	⟨ 0.0, 1 ⟩	⟨ 0.0, 1 ⟩	⟨ 0.0, 1 ⟩

Table 1: Diversity of optimal configurations: Each column represents the optimal memory configurations for a specific tiered-memory configuration.

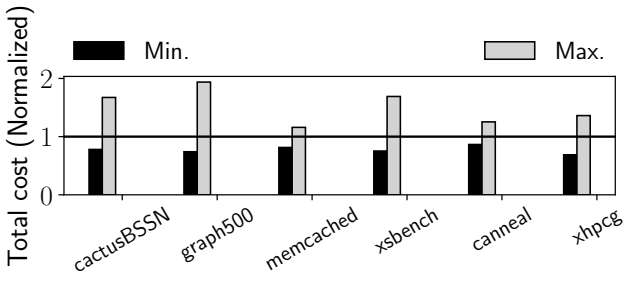


Figure 2: Costs for best/worst/average configuration, normalized to the average cost of all configurations.

Table 1 shows the optimal tiered memory ratio and LLC configuration across six different workloads and three memory technologies. Each column represents a specific tiered memory technology where the slow memory is s times slower and p times cheaper than DRAM. Performance per TCO optimal configurations for each workload are shown in the form of ⟨ slow memory ratio, LLC size ⟩. For example, if the slower memory tier has a 3× higher read latency but 0.4× lower cost, the cost-optimal configuration for the workload *graph500* has a ⟨ slow memory ratio = 0.4, LLC size = 28 ways ⟩. As we can see, there exists no configuration that is uniformly best for all workloads or memory technologies. Figure 2 further shows the minimum and maximum cost for the different hardware configurations and workloads (normalized to the average cost of all configurations). As can be seen, customers spend 1.2–1.5× less for the optimal configuration compared to the average configuration and 1.4 - 2.6× less compared to the worst configuration.

While this work mainly evaluates two memory tiers and variable LLC capacity, it can be easily extended to handle additional tiers and other resources such as memory bandwidth. We investigate general techniques that can be used to uncover application-specific properties for the workloads regardless of the underlying memory technology—the slow

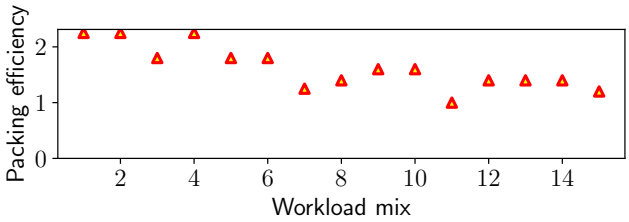


Figure 3: The packing efficiency improvement achieved by a resource-optimal policy over the naive policy.

memory tier can be locally attached 3DXP or CXL-attached DRAM/3DXP. Furthermore, while we do not explicitly discuss latency-critical workloads, we efficiently support them through customizable objective functions, *e.g.* TMC can find the most cost-effective configuration fulfilling a certain SLO requirement (<X% slowdown).

2.2 Resource Efficiency for Cloud Operators

Cloud customers request VM instances of a specific hardware configuration. The cloud's VM scheduler is responsible for selecting a server that can hold the new VM according to the hardware requirements and the current availability of machines in the cluster. One important aspect for optimizing the cost efficiency of such clouds is to optimize the packing density [70]. If VMs can be packed into fewer machines at a given time, idle machines can be powered down to save energy and cost, or they can be used to run low-priority batch jobs. Packing inefficiency leads to resource stranding where one of the resources (e.g. vCPUs) becomes fully utilized while others (e.g. memory) are not. Existing cluster schedulers [27, 29, 71] consider packing efficiency during job scheduling to maximize cloud resource utilization. However, they fail to be effective if the resource demand of the VM workload is fundamentally unbalanced. For example, if all workloads at a given moment request disproportionately large amounts of DRAM, a large amount of second-tier memory can be left unused. VM configurations need to adapt based on the real-time resource utilization of the cloud.

We thus investigate the potential upper-bound benefit of considering packing efficiency when choosing a tiered memory configuration. In particular, we determine the optimal slow-to-fast memory ratio based on resource availability. We evaluate 15 different workload mixes consisting of 4 applications each and compute the benefit provided by the resource-optimal policy considering packing efficiency over a naive policy. For additional information regarding the workload mix, please refer to Section 4. The naive policy requests optimal tiered memory allocations for each individual application in isolation, whereas the resource-optimal policy considers the availability of physical hardware resources. For instance, if a machine contains 3× more slow than fast memory, the resource-optimal policy reserves 0.5× fast and 1.5× slow memory regardless of what the actual optimal slow-to-fast memory ratio of an application is. As shown in Figure 3, the resource-optimal policy achieves up to 2.2× higher packing efficiency than the naive cost-optimal configuration, motivating the consideration of both configuration cost and packing efficiency.

2.3 Performing Optimal Resource Allocation

Resource allocation in tiered memory systems must be performed whenever an application is submitted to the cluster, considering the workload's requirements and currently available physical resources. Performing *optimal* allocations is generally unfeasible due to the large search space of configurations and resources that need to be explored. It is crucial to minimize the required time for identifying a good memory configuration for a workload since the primary objective of using tiered memory systems is to reduce the total cost of ownership (TCO). In particular, the runtime cost of determining a near-optimal configuration should be significantly lower than executing the workload. Prior works tried to address this challenge mainly through black-box machine learning techniques.

Machine learning. Machine learning techniques [52, 58, 67, 69] such as KNN, Support Vector Machines (SVM), or regression can predict application performance under a specific machine configuration. However, traditional machine learning methods are not aware of search cost considerations and often require a large number of samples.

Collaborative filtering. Collaborative filtering [20, 41] predicts the performance of an unknown application across various configurations using sparse performance data obtained from profiling training workloads. However, systems like Selecta [41] need substantial training on diverse workloads and configurations to achieve accurate predictions.

Bayesian Optimization. Previous works [3] propose the use of Bayesian optimization (BO) to minimize the search cost in optimizing an objective function that maps a specific configuration to an execution cost. BO treats the objective function as a black box and aims to find an optimal solution with minimal samples. However, incorporating resource efficiency goals with BO is challenging, as it requires initiating a new search every time resource utilization changes, leading to increased search costs.

3 TMC Design

In this section we describe how TMC produces tiered memory and LLC configurations optimizing cost efficiency for both the cloud provider and user.

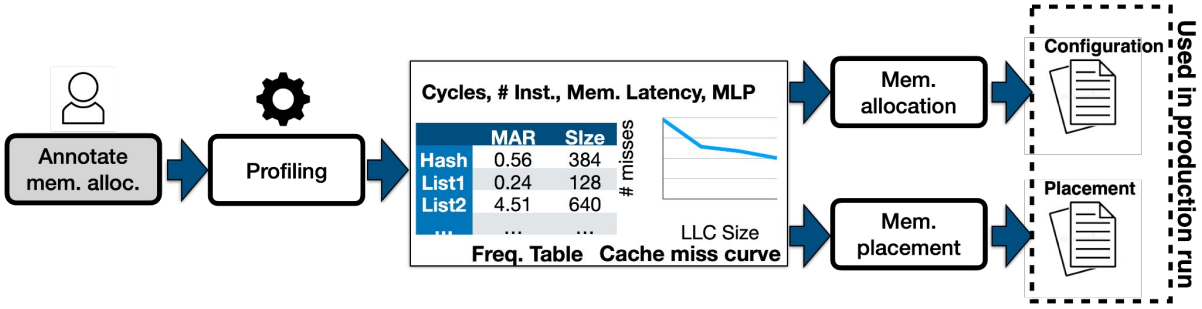


Figure 4: Workflow of TMC. The *frequency table* records the memory access rate (MAR) and size of each data structure, while the *cache miss curve* estimates the overall memory access rate by considering the number of allocated last-level cache (LLC) ways.

name	description
$fast_{max}$	0% slow memory, maximum LLC size
$fast_{min}$	0% slow memory, minimum LLC size
$slow_{max}$	100% slow memory, maximum LLC size

Table 2: Reference configurations

3.1 Overview

TMC requires 3 inputs: i) the workload submitted by a cloud customer, ii) the cost model stating the price of slow and fast memory as well as the cost of an LLC way, and iii) the latest resource utilization of the available hardware. To enable its performance predictions, TMC monitors the memory consumption of an application during a profiling stage. TMC then automatically determines two VM parameters: the near-optimal memory ratio of local DRAM and slow memory and the number of LLC-ways allocated to the VM.

In line with prior work [23, 74], we observe that the characteristics above are almost always data-structure specific. As a result, our proposed methodology determines application properties such as memory access rate on a per data-structure level and not on raw page data. One advantage of this design choice is that it takes advantage of application semantics, allowing for memory data access tracking at a coarser granularity. This results in reduced memory consumption for storing access statistics, lower sampling overhead, higher accuracy, and independence from the OS page size. For example, in our applications (as shown in Table 5), we have up to 30 data structures, which would require only 960 bytes of memory for storing access statistics, assuming a conservative overhead of 32 bytes per data structure. Additionally, while it is challenging to track the access rate for individual pages of 4 KiB or 2 MiB using existing techniques [1, 24, 51, 61] such as Intel's precise event-based sampling (PEBS), assessing the access rate at the data structure level is straight forward. Being agnostic to the OS page size is also important for applications that utilize huge pages [51].

To enable data-structure profiling, each memory allocation needs to be associated with a tag that links to a specific data structure. In line with prior work [23], our prototype offers a customized memory allocation interface enabling the user or an automated script can provide a tag. Memory allocated for distinct data structures (e.g., sharing the same tag) will be served from separate memory chunks of coarse-grained memory units of *e.g.* 2 MiB. Consequently, data from different data structures can be prevented from sharing the same page. The procedure for annotating data structures typically starts by pinpointing the source-code locations where substantial memory allocations occur. These allocations are then substituted with our customized memory allocator that includes an extra tag passed as an argument. To prevent overlooking any allocations, one can utilize an LD_PRELOAD wrapper to intercept all malloc calls and trace the call stack. Regarding containers, our prototype utilizes a customized allocator, guaranteeing that all allocations made by these containers receive appropriate tagging. In our experimental work, we find ourselves modifying an average of 23 lines of code for the benchmarks. In Section 3.6, we discuss an automated approach to assign tags, eliminating any burden to the programmer.

Figure 4 outlines the workflow of TMC. TMC initially measures important application properties, including the memory access rate, observed latency, and memory level parallelism (MLP), alongside the execution time of the workload under three reference configurations ($fast_{min}$, $fast_{max}$, and $slow_{max}$). Note that while our current approach selects only three reference configurations, incorporating additional reference points would enhance the model's accuracy, albeit with an associated increase in search cost. Detailed information about these reference configurations can be found in Table 2. In Section 3.2 and Section 3.3, we demonstrate how

	Definition
CPI	Cycle Per Instructions
CPI_{cache}	CPI in a system with perfect LLC cache
MAR	Memory access rate (accesses per instruction)
MAR_{slow}	Memory access rate to second-tier memory
MLP	Effective memory level parallelism
L_{dram}	Effective DRAM latency
L_{slow}	Effective slow-memory latency
ΔL	Latency increase in slow memory over fast memory

Table 3: Performance metric list with definitions.

TMC utilizes these application properties and the execution time of the reference configurations, to facilitate efficient and accurate performance estimation. An accurate performance estimation model allows TMC to select appropriate configurations for the tiered memory. For each configuration with different sizes of LLC, fast memory, and slow memory, we utilize Equation 1 to calculate the total cost. It should be noted that the cost of the virtual machine (VM) for each configuration is determined by the cloud's price profile. Further details about the price profile used in our experiment can be found in Section 4.1. Furthermore, in Section 3.4, we will showcase the application of the estimated performance profiles in guiding data placement in tiered memory systems. TMC generates data placement instructions based on data structure hotness, and a hybrid memory-aware allocator enforces per-data structure limits to optimize the efficiency of the tiered memory system.

3.2 TMC Performance Model

In this section, we analyze prior work on application performance modeling and then propose an improved model that can handle tiered memory systems. Our model is based on prior work [15] that proposed Eq. 2 to quantify the relationship between off-chip memory accesses and the application performance in a homogeneous system (*e.g.* DRAM-only). Table 3 shows the performance metrics used by the devised models.

$$CPI = CPI_{cache} + \frac{MAR \times L_{dram}}{MLP} \quad (2)$$

In Eq 2, CPI_{cache} represents the on-core CPU cycles, while $\frac{MAR \times L_{dram}}{MLP}$ accounts for the off-core cycles spent waiting for memory accesses. When memory accesses are processed sequentially, the off-core cycles are $MAR \times L_{dram}$ *e.g.* $MLP = 1$. However, due to existing memory level parallelism (MLP), some of the memory latency is amortized, allowing for concurrent memory accesses and reducing the effective off-core cycles. We expand upon Eq.2 to encompass systems featuring a tiered memory hierarchy. We incorporate the performance penalty introduced by a second-tier memory, as depicted in Eq.3:

$$CPI = CPI_{cache} + \frac{MAR \times L_{dram}}{MLP} + \frac{MAR_{slow} \times \Delta L}{MLP} \quad (3)$$

Equation 3 signifies that various factors influence the performance of applications in tiered memory systems. These factors include the on-core performance (CPI_{cache}), the memory access rate (MAR and MAR_{slow}), the sensitivity to memory latency (ΔL), and the presence of memory level parallelism (MLP).

- **On-core performance.** The on-core cycles (CPI_{cache}) exclusively account for the non-blocking CPU cycles, excluding the cycles spent waiting for memory stalls. The consumption of on-core cycles by an application depends on the specific CPU microarchitecture and the nature of the application, such as whether it is computation-intensive or memory-intensive.
- **Memory access rate.** The overall memory access rate (MAR) is influenced by the application's memory access pattern and the size of the CPU cache. In the presence of memory tiering, a portion of the memory access is fulfilled by the slower second-tier memory, depending on the size of the second-tier memory and the data structures placed within it.
- **Memory latency.** The effective memory latency can be highly dependent on the access pattern of the application [8, 11, 32, 35, 46, 79]. For instance, the queuing delay within the memory subsystem depends on the memory bandwidth, while the access patterns affect the probability of DRAM row hits. Thus, we assess the average end-to-end memory latency for each application individually.
- **Memory level parallelism.** Memory accesses incur significant latency and result in prolonged CPU stalls. To mitigate some of this latency, modern Out-of-Order CPUs execute multiple memory accesses concurrently. Memory level parallelism (MLP) denotes the average number of concurrent, outstanding memory accesses during the execution of a program. It is important to note that in this paper, we refer to MLP as effective MLP, which is determined not only by the application's structure (instruction parallelism) but also by the limitations imposed by the underlying memory system hardware (such as the instruction window's hardware constraints on parallelism).

3.3 Inferring Tiered-Memory Performance

We now describe the methodology of collecting application properties for informing TMC's performance model. We assume that performance properties such as on-core performance (CPI_{cache}), effective MLP, and effective first and second-tier memory latency (L_{dram} and L_{slow}) are not affected by the tiered memory configuration. This assumption is based on the observation that memory tiers differ on the

	CPU cycles	# Instructions	Memory Latency	MLP	Cache miss curve	Access frequency
Granularity	app.	app.	app.	app.	app.	struct.

Table 4: Application properties and their profiling level (application level or data structure level).

memory-system, but not CPU-core level, and has been validated through experiments. Table 4 provides an overview of the application properties collected during the profiling stage and outlines the specific granularity at which these performance properties are observed.

PMU Counters. Our TMC simulator in Section 4 emulates hardware performance monitoring unit (PMU) counters that measure on-core, non-blocking CPU cycles, MLP, and memory access latency. The on-core CPI and MLP are obtained by averaging measurements of the three reference configurations (Table 2). Additionally, the access latency (L_{dram}) of the first-tier memory can be measured in either the $fast_{max}$ or $fast_{min}$ reference configuration, while the slow memory's latency (L_{slow}) can be measured in $slow_{max}$. Contemporary CPUs offer the following hardware performance counters to capture the required application properties:

- **On-core cycles.** All X86 processors support PMU counters to measure the total CPU cycles (`CPU_CLK_UNHALTED.CORE`) and memory stalls (`CYCLE_ACTIVITY.STALLS_L3_MISS`) from which the on-core CPU cycles can be derived.
- **MLP and memory access latency.** Due to the lack of general performance counters in contemporary Intel CPUs to directly measure MLP and average memory latency, one approach is to indirectly assess them by computing the amortized performance penalty using the application properties collected in $fast_{max}$ and $slow_{max}$. Further details about this approach can be found in Section 5.

Memory access rate. The memory access rate (or LLC miss rate), non-linearly depends on the last-level cache (LLC) size and the spatial and temporal locality characteristics of an application. To estimate the memory access rate for different LLC sizes (number of ways), we utilize the concept of *cache miss curves* proposed by Qureshi [60]. These curves are derived from per-set hit counters using the LRU stack property. By measuring a cache miss curve in any of the three reference configurations, we can approximate the overall access rate (*MAR*) based on the number of allocated LLC ways. After determining the memory access rate, TMC needs to estimate the fraction of accesses that are served by the second-tier memory (MAR_{slow}). Therefore, we introduce the *frequency table*, which is constructed from application memory traces during the profiling step, by tracking the accesses and memory region for each data structure. Each memory access is mapped back to its corresponding data structure to update its access count. Data structure access frequencies can be obtained via Intel's Precise Event-Based Sampling (PEBS) and other similar hardware sampling techniques that enable the recording of LLC-miss addresses directly. We will investigate the implication of the PEBS sampling rate on accuracy and performance overheads in Section 5. The frequency table enables TMC to rank data structures according to their "hotness", which is used for determining data placement in the fast and slow memory tiers (Section 3.4). In particular, the slow-memory access rate is estimated as the cumulative access rate of data structures placed in the slow memory. In the example shown in Figure 4, assuming a working-set size of 2.5 GiB, when the slow memory ratio is 20% (512 MiB), the data structures `Hash` (384 MiB) and `List1` (128 MiB) will be placed in the slow memory. Therefore, the access rate to the second-tier memory can be estimated as the accumulated access rate of `Hash` and `List1` (*i.e.* 0.8).

LLC-specific memory access rate. To determine the optimal number of LLC-ways for an application, TMC needs to estimate the slow-memory access rate for a given LLC size. A naive approach would be to measure the slow-memory access rate under every configuration, collecting a *frequency table* for each LLC-size configuration introducing a significant profiling overhead. We address this challenge through an approximation methodology that requires only two *frequency tables* obtained when running $fast_{min}$ and $fast_{max}$. We define the *r% slow-memory miss curve* as a curve depicting how the slow-memory access rate changes over the LLC size with a slow memory ratio of *r%*. The approximation is based on the assumption that all *slow-memory miss curves* of an application are likely to have the same slope. For instance, the "knee" in the 50% *slow-memory miss curve* is expected to occur at the same position (*e.g.* LLC ways = 3) as in the *cache miss curve*. The process is detailed in Figure 5. First, for each slow memory ratio *r%*, the access rate to the second-tier memory is known only for ⟨ slow memory=r%, LLC=MIN ⟩ and ⟨ slow memory=r%, LLC=MAX ⟩, which constitute the start and end points of the *r% slow-memory miss curve*. To approximate the complete miss curve for the target ratio, we first align the *cache miss curve* with the start point of the *slow-memory miss curve* (step 2) and then scale it vertically to fit the endpoint (step 3).

3.4 Data Placement

While the presented methodology above enables TMC to determine a near-optimal slow-to-fast memory ratio and the number of LLC ways, we must also devise a policy to decide which data structures should be placed in fast, respectively,

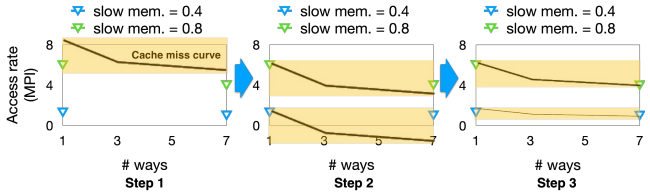


Figure 5: Estimating LLC-specific memory access rate with the *cache miss curve*. The approximation assumes that all slow-memory miss curves of an application exhibit a similar slope.

slow memory. TMC utilizes a policy based on the access count per MiB as determined by $\frac{Access\ Count}{Size}$ to represent the hotness of a data structure.

As illustrated in Figure 4, in addition to determining the tiered memory configuration, TMC generates data placement instructions that are utilized at runtime. These instructions take into account the optimal allocation between the first and second-tier memory, as well as the hotness of each data structure. Based on this information, we can determine the appropriate allocation of first-tier memory for each data structure. For example, in the case presented in Figure 4, assuming the determined optimal slow memory capacity is 256 MiB and fast memory capacity is 1024 MiB, TMC allocates 130 MiB of fast memory and 256 MiB of slow memory for the `Hash` data structure while placing all other data structures into fast memory since they are considered hotter. At runtime, a hybrid memory-aware TMC memory allocator enforces the per-data structure limit on DRAM capacity. This allocator ensures that the allocated memory for each data structure adheres to the defined limits optimizing the tiered memory system efficiency. In our present implementation, we utilize Jemalloc hooks to allocate data structures within distinct arenas. The differentiation is achieved through tags assigned either by programmers or the language runtime, as elaborated in Section 3.6.

While prior work such as X-MEM [23] has proposed more complex policies that consider MLP for data structure placement, we observed that such an approach only provides a few performance improvements over a policy based on access count. In addition, X-MEM introduces a 40× slowdown as it must use expensive application instrumentation to analyze continuous memory access traces in order to distinguish memory access patterns *e.g.* pointer chasing, sequential, or random.

3.5 Optimizing Packing Efficiency

The performance estimation model, as discussed in Section 3.2 and Section 3.3, serves the purpose of predicting the execution time for a given configuration. To determine the values of C_i and C_{min} for use in Algorithm 1, one can leverage the estimated execution time and unit cost associated with the specific configuration, as described in Equation 2. Algorithm 1 shows the pseudo-code of our configuration selection. Our selection algorithm consists of two rounds. In the first round, we identify all configurations that satisfy the cost-performance objective of the customer. If the estimated cost of a configuration is within a threshold `T` of the estimated optimal cost, we consider it as a cost-optimal. In the second round, we pick a configuration that maximizes the resource efficiency for the cloud provider. We propose a new heuristic, *packing penalty*, to evaluate the impact of a configuration on the resource efficiency in the cloud: a configuration with a lower packing penalty makes more efficient usage of cloud resources and vice versa.

The packing penalty is calculated as the dot product of the two vectors U and D. The vector U represents the overall resource utilization of each configuration, while the vector D represents the resource demand of each configuration. The dimensions of these vectors correspond to the number of resource types, which is four in our work (e.g., CPU, LLC, slow memory, fast memory). The rationale behind the packing penalty is to penalize configurations that heavily utilize scarce resources. For example, let's consider a simplified

Algorithm 1 Configuration Selection Algorithm

$Conf_i$ ▷ The i'th candidate configuration
C_i ▷ Total execution cost for $conf_i$
C_{min} ▷ Minimum execution cost (estimated)
R ▷ Total resource capacity in a machine
U ▷ Overall resource utilization in a cloud
N ▷ Number of candidate configurations
T ▷ Acceptable cost deviation from optimal

procedure OPTIMIZATION(...)
  ▷ First round: optimize the cost for the customers
  for each $i \in \{1...N\}$ **do**
    if $\frac{C_i}{C_{min}} - 1 \leq T$ **then**
      opt.insert($Conf_i$)
    end if
  end for

  ▷ Second round: optimize the resource efficiency
  for each $i \in \{1...S\}$ **do** ▷ S is the size of opt
    $D_i \leftarrow \{\frac{opt_{i,cpu}}{R_{cpu}}, \frac{opt_{i,dram}}{R_{dram}}, \frac{opt_{i,slow}}{R_{slow}}, \frac{opt_{i,llc}}{R_{llc}}\}$
    $penalty_i = D_i \cdot U$
  end for
  Sort the opt according the $penalty$ (increase)
  return opt_0
end procedure

scenario with two resource types: fast memory and slow memory. We have identified two cost-optimal candidate configurations: *Config_1* requires 0.2 GiB of fast memory and 0.8 GiB of slow memory, while *Config_2* requires 0.8 GiB of fast memory and 0.2 GiB of slow memory. Suppose the fast memory is under high pressure (*e.g.* 70% utilization), while the slow memory is relatively idle (*e.g.* 20% utilization). In this case, we would prefer *Config_1* due to its lower packing penalty ($0.2 \times 0.7 + 0.8 \times 0.2 = 0.3$) compared to *Config_2* ($0.8 \times 0.7 + 0.2 \times 0.2 = 0.6$).

3.6 Discussion

Application transparency. We now explore several options to alleviate the adoption of the TMC methodology. First, TMC can by implemented as part of the language runtime, eliminating the need for developers to annotate memory allocations manually. In modern C++, heap memory objects are typically managed using smart pointers (*e.g.* `std::make_unique<T>`) or STL containers (*e.g.* `std::vector<T>`). Therefore, we can delegate the tagging responsibility to the language runtime. For example, the STL container can utilize the type information `T` as the tag for its memory allocation. Alternatively, we can allow the memory allocator to dynamically generate a tag at runtime by examining the call stack. This approach leverages the fact that the same memory allocation call site always allocates memory objects of the same type.

Bandwidth. Our prediction model assumes that the end-to-end memory latency of an application is relatively stable across different configurations. We found that this assumption holds as long as the memory bandwidth of the system is not overly saturated, particularly less than 80% of the peak. As demonstrated in prior studies [25, 35, 36, 79], there is a "knee" in the bandwidth-latency curve at around 80% of the maximal bandwidth. For most of the operating range, memory latency is relatively flat, however, it increases exponentially after the "knee". As reported in [35], the "knee" for 3DXP when dealing with random read workload is at around 10 GiB/s where the latency increases from 300ns to 400ns. Our technique assumes that the cloud provider enforces workload mixes via scheduling that consumes at most 80% of a machine's DRAM bandwidth. This has been naturally the case for all workload mixes evaluated in this paper. Google has reported [37] that data center applications are almost exclusively DRAM latency and not bandwidth limited.

Tail latency. While TMC predicts latency (execution time), it currently does not allow predicting tail latency. We assume that other tail-latency mitigating techniques are deployed [19] and as a result, the application is not too tail-latency sensitive. This approach should apply to sufficient application areas, as Google has reported that high-priority workloads run on dedicated and not shared machines [7]. Prior works predicting optimal configurations also do not consider tail latency [3, 41].

Application specific MLP. Prior work has proposed measuring data structure specific MLP by extending the memory controller [46] or by adding expensive instrumentation [23]. However, our study indicates that replacing application specific MLP with data structure specific MLP provides little improvement over the prediction accuracy (less than 1%). Hence, our work measures the MLP of the application, assuming MLP is identical across different memory configurations. Our approach has shown to be accurate when applied in real systems for predicting application performance (Section 5).

Noisy profiles. If the profiling runs of an application happen to be scheduled on a machine that is under abnormal conditions *e.g.* overloaded, TMC might produce a performance model that is not representative of the machines in the cluster. To overcome this problem, we can design a micro-benchmark that performs operations exercising different resources in the system, and that is executed periodically in the machines of the cluster, reporting back the representative metrics. We will schedule the profiling runs on a machine whose condition is consistent with the common machines in the cluster.

Dynamic tiering. In data centers, applications are recompiled, deployed, and profiled multiple times a day, enabling TMC to adapt to changing inputs, application characteristics, and hardware resources. Google [13] has shown that input characteristics change slowly and that profiles only become outdated over weeks. TMC can be deployed as an always-on system that constantly profiles and re-allocates memory for new application invocations based on the existing hardware resources. TMC provides accurate predictions with low PEBS sampling overhead as demonstrated in Section 5. In such a dynamic environment, the improved search time provided by TMC over prior work is particularly important.

Practicality in Cloud Deployments. In private clouds PEBS access is not an issue. Linux supports PMU virtualization with vPMU. AWS supports the virtualization of core-PMUs. Finally, Clients can also pre-profile their workloads on native machines and then submit the information to the cloud operator.

4 Evaluation in Simulation

In this section, we first present our experiment methodology. We then analyze the effectiveness of TMC.

4.1 Experimental setup

Cloud VM simulation. We simulate physical hardware in the cloud using the cycle-accurate simulator Scarab [33]. We modify Scarab to implement the required performance

	# Structs.	Description
CactusBSSN [12]	30	Model a vacuum flat space-time
Graph500 [55]	5	BFS search on an undirected graph
Memcached [17]	3	Workload C in YCSB
XSBench [68]	5	Monte Carlo neutron transport
Canneal [10]	10	Opt. routing cost of a chip design
XHPCG [21]	13	Preconditioned conjugate gradient

Table 5: Description of the benchmarks including the required number of annotated data structures (*i.e.* tags) for each application.

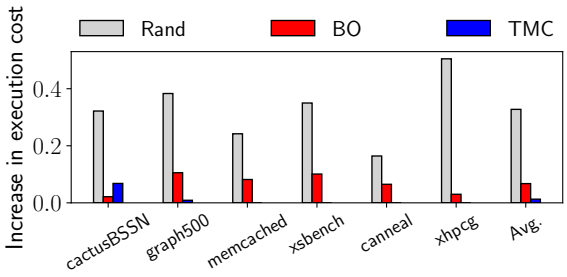


Figure 6: Execution cost increase over exhaustive search

counters for capturing cache miss curves and application-specific properties (see Section 3.3) required by TMC. The simulated machines in our study contain 12 cores, 12 MiB 48-way associative LLC and 24 GB DRAM. Existing machines support 16-20 LLC ways which is insufficient for realizing optimal configurations. This motivates us to evaluate a larger number of LLC ways to enable fine-grained LLC allocation. Alternatively, future systems can also employ more advanced, fine-grain cache partitioning techniques such as vantage [65]. Although our work makes no assumption on the type of media that will be used in the slow memory tier, we simulate the performance characteristics of 3DXP memory by default *i.e.* 3× [35, 79] the DRAM latency (100 ns). In our experiment, the slower memory tier is provisioned by attaching a 128 GiB 3DXP DIMM to a server, except for Section 4.5 where we explore the effectiveness of TMC in other memory tiering architectures. In particular, we simulate a fast memory tier (DRAM) and a CXL-attached memory pool as the slow tier. Our CXL-based disaggregated system provides a shared memory pool for each 8-node rack. Similar to prior work [45], we add an additional latency of 85 ns to each CXL access *i.e.* the end-to-end memory latency is comprised of the CXL delay and the access latency of the second tier memory. New allocated VMs are added to a queue in order of their arrival. Every time a new virtual machine is created, the scheduler checks all machines to find one with sufficient resources. TMC analyzes the machines in random order, and places the job on the first one that has the required available resources.

IaaS cloud. We assume that customers can choose a slow memory ratio out of 11 slow memory ratios (0%, 10%, 20%, ... , 100%) and an LLC size out of seven configurations (1, 2, 3, 4, 12, 20, 28) utilizing technology such as Intel's cache allocation technology (CAT) [34, 65]. In total, there exist 77 candidate configurations. We obtain the hourly cost for a single vCPU and 1 GB of DRAM by using the least square method to solve a system of equations derived from all VM instances in the Msv2-series of Microsoft Azure. We assume the cost of 3DXP memory is 0.4× that of DRAM. Following the methodology described in [83], we obtain the hourly cost for a unit of LLC capacity according to the estimated area percentage of the LLC in a CPU chip.

Workloads. The workloads used in our experiments cover a broad spectrum of applications. The specifics of these workloads are available in Table 5. We modify the applications to utilize our custom memory allocator that assigns each data structure with a tag. For each application, we performed a basic-block vector-based analysis (Simpoint-like), determining that 2 billion instructions are representative in regards to IPC, MPKI, branches, and other metrics. We pick four out of the six workloads to form a workload mix resulting in 15 workload mixes in total. We believe four workloads strike a good balance between mix diversity and the total number of workloads in a mix. A workload mix represents the workloads that will be submitted to the cluster in an experiment. We select one of the workloads of the mix as the newly submitted workload. After a new workload arrives (randomly selected from the workload mix), TMC finds an appropriate VM configuration for the new workload. The new VM request is submitted to the cluster scheduler. After completing a certain number of workloads (5000 in our evaluation), we report the average execution cost for all workloads submitted to the cloud.

Baselines. We compare TMC with the following strategies: i) Exhaustive search (ES) finds cost-optimal configurations by running all the configurations. It provides an upper bound on the overall performance. ii) Random (Rand) selects a configuration randomly from a set of candidate configurations without any test runs. iii) Bayesian Optimization (BO) is a state-of-the-art solution that has been used in prior work [3] to reduce the number of samples to reach a cost-optimal configuration. In our experiment, we set the Expected Improvement (EI) to 5% and use three initial samples.

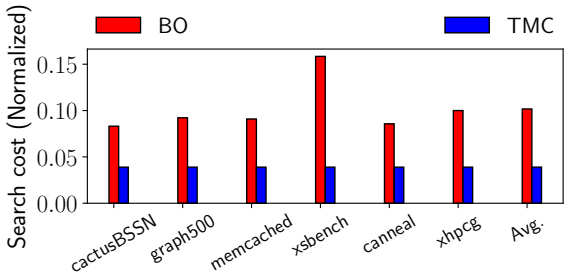


Figure 7: Search cost of TMC and BO, normalized to the search cost of exhaustive search (ES). ES and Rand are omitted as they are one and zero.

4.2 Execution and Search Cost

In this work we aim to learn cost-optimal tiered-memory configurations with minimal search overheads. For each workload in Table 5, we utilize TMC and the baseline techniques to recommend a VM configuration and the run the workload with the recommended configuration. We repeat the experiment 10 times and report the average number of test runs (search cost) and execution cost. Figure 6 shows the execution cost of the VM, LLC, fast and slow memory configuration determined by TMC and the baseline techniques. As can be seen, TMC reduces the execution cost by 1.3× over Rand and by 1.05× over BO in average. TMC only incurs a 2% higher TCO per performance than the ES's cost-optimal configuration. While ES provides the best performance in terms of cost minimization, it also incurs the highest search cost of all approaches as shown in Figure 7. For ES, every configuration needs to be run at least once. For instance, if an application has a one-minute average execution time and there are 77 candidate configurations in our experimental setup, it would require 77 minutes of profiling. Our TMC provides 26× lower search cost than ES and 3× lower search cost than BO. Rand imposes no search overhead by simply choosing a configuration randomly, however, it has no way of controlling the quality of the recommended configuration, and, as a result, it increases TCO by up to 50% and 33% on average compared to ES.

4.3 Improving Packing Efficiency

The second goal of TMC is to increase packing efficiency for the cloud operator without introducing a significant cost increase for the customer. For the cost increase, we set the threshold T to 2.5% so that the recommended configuration can be at most 2.5% more costly than the optimal configuration. The compact cluster size [70] refers to the minimum cluster size required to accommodate a given workload. In our work, we utilize the compact cluster size as a metric to assess the packing efficiency of the cloud. We determine the compact cluster size by progressively reducing the cluster size until the job is rejected due to insufficient available machines. A smaller compact size indicates a higher level of packing efficiency, as it indicates fewer machine resources are required to sustain the workload.

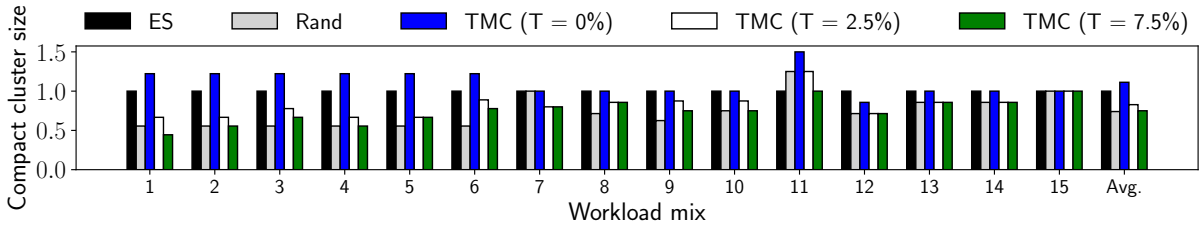
Figure 8 compares the compact cluster size and the average running cost achieved by the evaluated schemes. Since the Bayesian Optimization (BO) methodology does not prioritize resource efficiency and only tries to optimize the search cost, we do not include it in this comparison. The compact cluster size and the average running cost are normalized to the ES configuration. As can be seen in Section 4.2, exhaustive search recommends actual cost-optimal configurations, however, it entails significant search overheads. As compared to ES, TMC reduces the compact cluster size by 17% and introduces only a 1.5% higher cost. In addition, we also observe that TMC can indeed control the quality of the recommended configurations and achieve a significantly lower execution cost compared to a randomly selected configuration.

4.4 Threshold Sensitivity Study

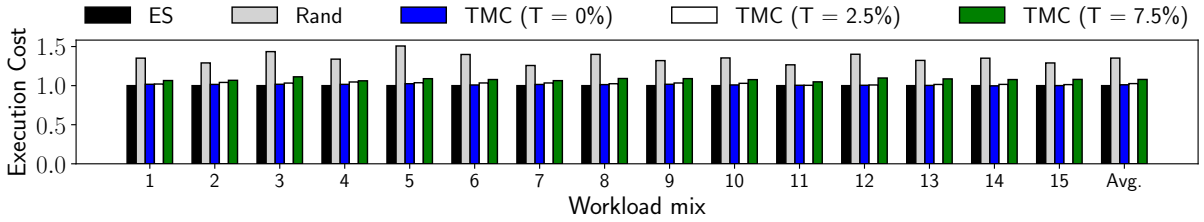
We next study the impact of the threshold T. A larger threshold T allows more configurations to be deemed cost-optimal, enabling TMC to increase further resource efficiency. Additional details about the threshold T can be found in Section 3.5. For example, when the T increases from 0% to 2.5% (resp., 7.5%), TMC can boost resource efficiency by reducing the compact cluster size by 25% (resp., 32%) as shown in Figure 8. On the other hand, increasing T also lowers the stand overall performance of the cost-optimal configurations, increasing the execution cost. For example, the average cost execution of the configurations recommended by the TMC is 1.1%, 2.6%, and 7.9% higher than the optimal cost when the threshold T is 0%, 2.5% and 7.5% respectively.

4.5 Memory Tiering Sensitivity Analysis

This experiment investigates the effectiveness of TMC under various tiered memory architectures. The two *Local* configurations deploy slow memory locally via DIMMs: *128L* attaches one 128 GiB 3DXP module and *256L* attaches two 128 GiB modules to each node. The current 3DXP modules only come in capacities of 128GB, 256GB and 512GB. As a result, *Local* can only increase the memory capacity at a very coarse granularity. The four *Pool* configurations explore rack-scale pooled deployment enabled by the emerging CXL technology to connect slow memories to 8 nodes via a CXL fabric: *128P* / *384P* / *640P* / *896P* respectively provide 128 / 384 / 640 / 896 GiB of 3DXP memory in the pool. For each such *far memory* configuration, we evaluate two configurations of local DRAM with either 24 GiB or 48 GiB per machine.



(a) compact cluster size



(b) execution cost

Figure 8: Efficiency of TMC's configuration selection (normalized to ES). TMC increases the efficiency while minimizing the cost penalty for the customer.

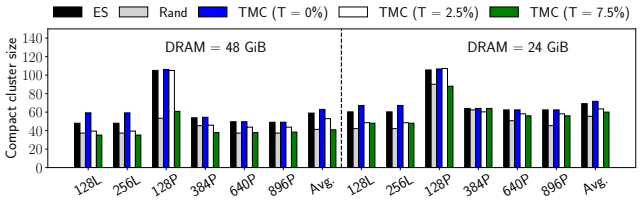


Figure 9: Effectiveness of TMC under various configurations of tiered memory.

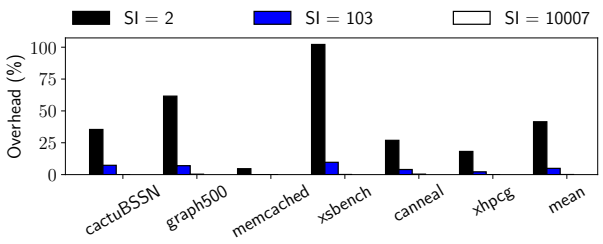
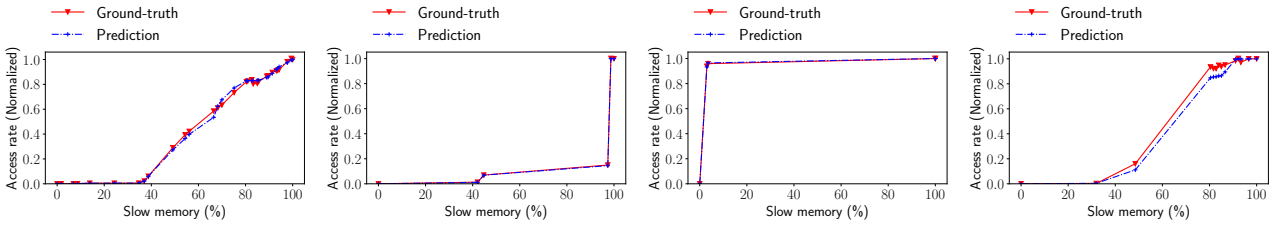


Figure 10: PEBS sampling overhead. We choose prime SIs to avoid bias from periodicities like prior work [49].

We run the 15 workload mixes under various tiered memory configurations to observe the average compact cluster size of each configuration shown in Figure 9. First, we demonstrate that TMC improves resource efficiency in nearly all architectures as compared to other approaches. For example, TMC reduces the compact cluster size on average by 30% (resp. 13%) as compared to ES when the DRAM size per machine is 48 GiB (resp. 24 GiB). Second, we observe that the *384P* and *640P* already allow TMC to achieve optimal resource efficiency when the size of the local DRAM is 24 GiB and 48 GiB respectively. *384P* and *640P* effectively provision 48 GiB and 80 GiB far memory per node. On the other hand, *Local* only allows the memory to be expanded at the coarser 128 GiB granularity, potentially leading to the over-provisioning and stranding of slow memory. Finally, data center operators must comprehensively consider both platform cost and resource efficiency in order to find an optimal server configuration. We particularly note two interesting examples: i) Doubling the local DRAM from 24 GiB to 48 GiB reduces the compact cluster size for all approaches; However, it also introduces significantly higher per-machine costs, ii) *640P* reduces platform cost by requiring 37.5% less 3DXP memory as compared to *128L*; however, TMC (T=7.5%) achieves 14% lower resource efficiency in a *640P*.

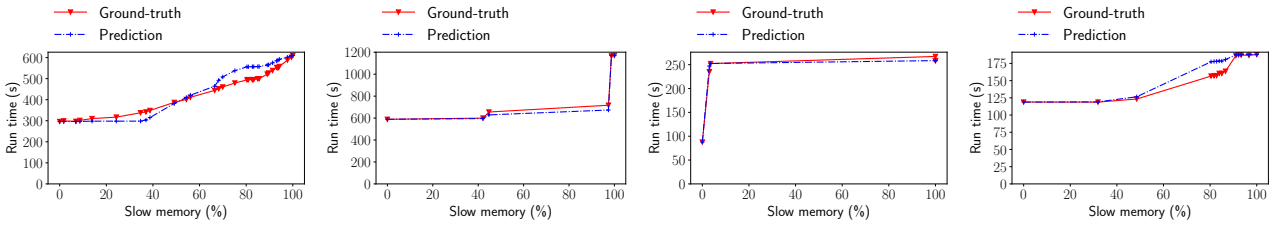
5 Real System Experiments

Simulation allows us to easily observe any application properties and thus enables us to quickly verify our proposed techniques on the performance estimation and the configuration selection. In this section, we introduce a proof-of-concept implementation and partly verify the applicability of using TMC in real systems. In particular, we build a working prototype that can predict how the run time changes



(a) cactusBSSN (b) graph500 (c) xsbench (d) xhpcg

Figure 11: Accuracy of estimating the slow memory access frequency in a real system using profiling.



(a) cactusBSSN (b) graph500 (c) xsbench (d) xhpcg

Figure 12: Accuracy of estimating the execution performance in a real system using profiling.

with the size of the slow memory. As hardware monitors for learning the cache miss curve [60] become available, TMC can be completely implemented in software. In the following section, we consider the following three main questions:

- What is the overhead of PEBS sampling?
- Can we accurately estimate the access frequencies?
- Can we accurately estimate the performance impact?

We utilize precise event-based sampling (PEBS) to intercept samples of memory accesses to estimate the access frequency of data structures. PEBS captures a snapshot of the processor state upon certain configurable hardware events. We program PEBS to monitor `MEM_LOAD_RETIRED.L3_MISS` events. PEBS can be configured with a sampling interval (SI). For a sampling interval of n, PEBS captures every n^{th} event into a buffer. When the PEBS buffer is full, an interrupt is triggered, during which TMC records the CPU state in a software-accessible buffer. We only record the virtual address accessed by CPU misses. The sampled memory accesses are then written to a file in a separate thread. As discussed in Section 3.3, we count the sampled accesses to different data structures in the offline analysis. We multiply the number of sampled memory accesses by the sampling interval (n) to estimate the actual access frequencies.

Due to the lack of general, well-documented performance counters that allow us to measure the MLP as well as the average memory latency in contemporary Intel CPUs, our current implementation measures MLP and the latency sensitivity of an application indirectly. In particular, we measure the amortized performance penalty introduced by accessing the slow memory. The amortized performance penalty takes into consideration both the latency-sensitivity as well as the effect of memory level parallelism on an application. We measure the CPI of an application in the DRAM-only configuration (CPI_{dram}), CPI in the configuration where all data is placed in the slow memory (CPI_{slow}), as well as the number of accesses to the slow memory (MPI_{slow}). The amortized performance penalty can then be computed via Eq. 4. The performance impact associated with placing data to the slow memory can be estimated by simply multiplying the access rate to the slow memory (estimated) and the amortized performance penalty.

$$Perf\ penalty = \frac{\Delta L}{MLP} = \frac{CPI_{slow} - CPI_{dram}}{MPI_{slow}} \quad (4)$$

5.1 Evaluation

Setup. To evaluate the proposed scheme, we use a server equipped with a Xeon Gold 5218 processor and a tiered memory hierarchy with six 32 GiB DRAM DIMMs (192 GiB in total) and a 128 GiB intel DC Persistent Memory.

Overhead. We first study the overhead introduced by PEBS which is important as it impacts the search cost of TMC. Figure 10 shows the sampling overhead for different

SIs compared to the application execution time without PEBS monitoring. The PEBS sampling overhead is comprised of induced pipeline flushes due to the PEBS assist, and the overhead for handling extra interrupts [2]. High sampling rates results in substantial performance overhead. With a SI of two, the performance overhead can be as high as 102.1% (41.4% on average). We configure PEBS to use a large sampling interval (10007). With such a large SI, we observe virtually no overhead (< 1%) due to PEBS sampling across all workloads.

Access-rate estimation. In our experiment, we move data structures of an application to the slow memory tier one by one in a random order and then measure the ground-truth number of accesses to the slow memory and the ground-truth run-time. Figure 11 shows the number of accesses to slow memory depending on the amount of data allocated in slow memory. We show the ground-truth and estimated number of accesses to the slow memory. As we can see, we achieve high accuracy in estimating the access frequencies even at a relatively low sampling rate (SI = 10007). The inaccuracies in the workload *xhpcg* is likely caused by the shadow effect of PEBS [80].

Performance estimation. Figure 12 shows the ground-truth and estimated run-time depending on the amount of application data allocated in slow memory. Overall, TMC achieves high accuracy in estimating the amortized performance penalty of different applications and, as a result, is capable of producing accurate predictions on the execution time of the application. This observation demonstrates that our assumption on the MLP and the memory latency as stated in Section 3.2 holds in most of the cases. However, we also observe that there is a relatively high prediction error (7%) on the performance although TMC achieves high accuracy in estimating the access rate for the *cactusBSSN* workload as shown in Figure 11a. The *cactusBSSN* workload presents an interesting example where the assumption that MLP / memory latency is identical across different memory configurations might not hold. However, we argue that adding new hardware [46] or using costly instrumentation [23] (40× slowdown) to capture the data structure specific MLP just for improving the accuracy for a few applications is not justified.

6 Related Work

Data tiering solutions. Hybrid memory systems have been proposed to allocate performance-sensitive data in first-tier memory and performance-insensitive data in second-tier memory, seeking to maintain performance at a lower cost. Given a fixed allocation between the first and second tier memory, prior work [1, 14, 23, 38, 39, 42, 46, 77, 78] can determine the best memory type for a given data item. While these prior works can provide raw performance guarantees, they are insufficient in the cloud setting, where operators and customers need to consider optimal cost efficiency as well.

Prior work on online data migration is complementary to our work. Meta's TPP [50] distinguishes between anonymous and file-backed allocations, preferring to place file-backed allocations in the slow tier. Increasingly, researchers [43, 50, 76] try to implement a proactive demotion approach that achieves two goals: the first is to always have free memory in the hot tier for hot pages by choosing separate high water mark and low water mark for the hot tier; the second is to use machine learning to guide proactive freeing of that memory [43, 45].

Memory profiling methodology. Previous research [1, 24, 51, 61, 72, 73, 78] has investigated different techniques for capturing application access patterns. Linux's page management approach [72, 73, 78] utilizes the hardware access bit to distinguish between hot and cold pages, aiding page replacement decisions. However, relying solely on scanning the accessed bit cannot accurately estimate the access rate of pages. It can only indicate recent access without counting the number of accesses. Frequent analysis of the access bit requiring TLB shootdowns is infeasible due to the performance overhead.

Another approach to assess the data access frequency is page sampling and poisoning, employed by works including Thermostat [1], TPP [51] and AutoNUMA [62]. These works utilize TLB misses as a proxy for memory accesses, which can introduce substantial inaccuracies for measuring the access rate.

Application [23] instrumentation represents another approach used for memory tracing and determining the access frequencies of different data structures. However, this method introduces substantial profiling overheads, resulting in a slowdown of up to 40 times. Additionally, it fails to account for the memory hierarchy, in particular, it is unaware of the filtering effect of CPU caches.

PEBS has been considered an effective option for tracking hot pages and has been explored in works such as HeMem [61] and TMTS [24]. However, our work differs in focus. While those works aim to select hot pages, our goal is to accurately measure the access rate. TMTS recognizes that some hot pages may be missing from PEBS sampling and resorts to page scanning as a last resort. This is not necessary in our approach. By leveraging application semantics, our method tracks memory access at a coarse granularity (data structure level), allowing for an accurate assessment of the access rate with minimal overhead.

Tiered-memory in data centers. Some data center operators [43, 76] have chosen to implement the slow memory tier using compressed DRAM or RDMA [4, 30, 59, 63] instead of new, directly-accessible memory technologies, such as 3DXP [56, 57] and CXL, considered in this paper. Linux

Zswap and related mechanisms compress swapped pages but are not fast enough for load/store-based interfaces. Other previous works [1, 50] dynamically tune the capacity of different memory tiers to improve cost efficiency while maintaining the service level objective. However, they offer no general solution on how to estimate the run time for a memory configuration. Mnemo [22] is a memory sizing tool designed only for key-value stores such as Memcached, Redis, and DynamoDB and cannot be used with general applications. HNVM [82] employs a hybrid of fast NVM (BBNVM) and slow NVM (PCM) as a persistent, compute-side cache. It optimizes the fast-NVM & slow-NVM ratio for an application. Unlike our work, HNVM utilizes costly exhaustive searches to build application performance profiles. In the context of computational memory, Sidekick [44] leverages Genetic Algorithms to optimize the placement of computations, tailoring placement decisions for each function invocation context individually.

Selecting cloud configurations. Machine learning techniques [52, 58, 67, 69] such as KNN, Support Vector Machines (SVM), or regression can be used to predict application performance under a certain machine configuration. However, traditional machine learning is not conscious of the search cost and may require a large number of samples. Ernest [69] has investigated reducing the amount of training data with experimental experiment design. However, the usage of Ernest is limited as it can only be used to predict the optimal number of instances for analytic workloads. CherryPick [3] relies on Bayesian optimization to reduce the number of samples that are necessary to reach a cost-optimal configuration. Our work instead utilizes the application-specific properties, and, as a result, minimizes the number of test runs required for performance estimation. While Selecta [41] focuses on selecting storage configurations, our work investigates how to select memory configurations for the cloud with heterogeneous memories.

Data center scheduling and resource allocation. Prior work on cloud scheduling is complementary to our research. Existing work on cloud schedulers [26, 27, 29, 66, 71, 81] typically tries to increase the resource efficiency using heuristics; However, optimizations on the scheduler can be useless if the VM workload as determined by the selection of machine configurations is fundamentally unbalanced. Optimizations [6, 18, 75] such as resource harvesting and oversubscribing that further improve resource utilization are orthogonal to our work.

7 Conclusions

This work investigates how to quickly identify ideal memory configurations for applications in tiered-memory cloud systems. TMC captures application-specific properties with existing performance monitoring hardware and uses them for accurate performance prediction. We demonstrate that TMC reduces the search cost by up to 4× while recommending high-quality configurations. Our approach additionally improves resource efficiency by 17% on average versus a naive policy that requests optimal allocations for each application in isolation. As a result, TMC provides the tools to efficiently support emerging tiered memory systems and to reap both performance and cost benefits.

Acknowledgments

We extend our gratitude to our shepherd Baptiste Lepers and the anonymous reviewers for their valuable insights and improvement suggestions. This work was generously supported by Samsung and NSF grants CCF-1942754 and CNS-1841545.

References

[1] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 631–644.

[2] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. 1–8.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI '17)*. 469–482.

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. 1–16.

[5] Amazon. [n. d.]. Amazon elastic compute cloud. https://aws.amazon.com/ec2.

[6] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. 2020. Providing *SLOs* for Resource-Harvesting *VMs* in Cloud Platforms. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI '20)*. 735–751.

[7] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *Proceedings of the 24th Int'l Symposium on High-Performance Computer Architecture (HPCA-24)*. IEEE, 643–656.

[8] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.

[9] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473.

[10] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph. D. Dissertation. Princeton University.

[11] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.

[12] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[13] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 12–23.

[14] Chia Chen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2014. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–12.

[15] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 213–224.

[16] Compute Express Link Consortium. 2020. Compute Express Link: The breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. 153–167.

[19] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[20] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

[21] Jack Dongarra, Piotr Luszczek, and M Heroux. 2013. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752* (2013).

[22] Thaleia Dimitra Doudali and Ada Gavrilovska. 2019. Mnemo: Boosting memory cost efficiency in hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 412–421.

[23] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. 1–16.

[24] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 2023 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 727–741.

[25] Sadagopan Srinivasan Li Zhao Brinda Ganesh, Bruce Jacob, and Mike Espig Ravi Iyer. 2009. CMP Memory Modeling: How Much Does Accuracy Matter?. In *Fifth Annual Workshop on Modeling, Benchmarking and Simulation*. 24–33.

[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*.

[27] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. 99–115.

[28] Google. [n. d.]. Create a VM with a custom machine type. https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type.

[29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI '17)*. 649–667.

[31] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces. In *Proceedings of the 27th International Workshop on Quality of Service (IWQoS '19)*. IEEE, 1–10.

[32] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[33] HPS. 2020. scarab. https://github.com/hpsresearchgroup/scarab.

[34] Intel. 2016. Introduction to cache allocation technology in the intel xeon processor e5 v4 family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[35] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[36] Bruce Jacob. 2009. The memory system: you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture* 4, 1 (2009), 1–77.

[37] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[38] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based {Key-Value} Stores Using Storage Class Memory as a Volatile Memory Extension. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 821–837.

[39] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 715–728.

[40] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash == local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

[41] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 759–773.

[42] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. Pageseer: Using page walks to trigger page swaps in hybrid memory systems. In *Proceedings of the 25th Int'l Symposium on High-Performance Computer Architecture (HPCA-25)*. IEEE, 596–608.

[43] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 317–330.

[44] Sanghoon Lee, Jongho Park, Minho Ha, Byung Il Koh, Kyoung Park, and Yeseong Kim. 2023. Sidekick: Near Data Processing for Clustering Enhanced by Automatic Memory Disaggregation. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[45] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. 2022. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241* (2022).

[46] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-based hybrid memory management. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing*. IEEE, 152–165.

[47] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th annual international symposium on Computer architecture*. 267–278.

[48] Heiner Litz, Maximilian Thuermer, and Ulrich Bruening. 2010. TC-Cluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect. In *2010 IEEE International Conference on Cluster Computing*. IEEE, 9–18.

[49] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. 2016. Laser: Light, accurate sharing detection and repair. In *Proceedings of the 22th Int'l Symposium on High-Performance Computer Architecture (HPCA-22)*. IEEE, 261–273.

[50] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *arXiv preprint arXiv:2206.02878* (2022).

[51] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 2023 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 742–755.

[52] Andréa Matsunaga and José AB Fortes. 2010. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 495–504.

[53] Microsoft. [n. d.]. Microsoft Azure: Cloud Computing Services. https://azure.microsoft.com/.

[54] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2021. *Introduction to linear regression analysis*. John Wiley & Sons.

[55] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[56] Yuanjiang Ni and Shuo Chen. 2020. Closing the performance gap between dram and pm for in-memory index structures. *Technical report* (2020).

[57] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L Miller. 2019. SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 836–848.

[58] Oliver Niehorster, Alexander Krieger, Jens Simon, and Andre Brinkmann. 2011. Autonomic resource management with support vector machines. In *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, 157–164.

[59] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit:{Low-Latency},{High-Throughput}, and Transparent Remote Memory via {Feedback-Directed} Asynchrony. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI '23)*. 181–198.

[60] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[61] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*. 392–407.

[62] Red Hat, Inc. 2012. AutoNUMA. https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.

[63] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI '20)*. 315–332.

[64] Samsung. 2015. Samsung Unveils Industry-First Memory Module Incorporating New CXL Interconnect Standard. https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard.

[65] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Int'l Symposium on Computer Architecture*. 57–68.

[66] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)*. 351–364.

[67] Akbar Sharifi, Shekhar Srikantaiah, Asit K Mishra, Mahmut Kandemir, and Chita R Das. 2011. METE: meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the 2011 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 13–24.

[68] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[69] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for {Large-Scale} Advanced Analytics. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '16)*. 363–378.

[70] Abhishek Verma, Madhukar Korupolu, and John Wilkes. 2014. Evaluating job packing in warehouse-scale computing. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing*. IEEE, 48–56.

[71] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*. 1–17.

[72] Vladimir Davydov. 2015. idle memory tracking. https://lwn.net/Articles/643578/.

[73] Carl A Waldspurger. 2002. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 181–194.

[74] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362.

[75] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*. 1–16.

[76] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 2022 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 609–621.

[77] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*. 1–14.

[78] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 331–345.

[79] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '20)*. 169–182.

[80] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 98–105.

[81] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. 265–278.

[82] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, Anirudh Badam, and David Wentzlaff. 2017. HNVM: Hybrid NVM enabled datacenter design and optimization. *Microsoft Research TR* (2017).

[83] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. 2016. CASH: Supporting IaaS Customers with a Sub-core Configurable Architecture. In *Proceedings of the 43th Int'l Symposium on Computer Architecture*. 682–694.

[84] Yanqi Zhou and David Wentzlaff. 2014. The sharing architecture: sub-core configurability for IaaS clouds. In *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 559–574.